

7. ООП на прототипах. Реализация в JavaScript

ООП на прототипах

Объекты создаются как копии или расширения других объектов (прототипов) , а не через строгую иерархию классов.

Это альтернативный подход к традиционному ООП, где классы определяют структуру и поведение объектов, а наследование реализуется через иерархию классов.

Особенности ООП на прототипах

1. Прототип как шаблон

Каждый объект может выступать в роли прототипа для другого объекта. Новый объект наследует свойства и методы прототипа, но при этом может их переопределять или расширять.

2. Динамическое наследование

Вместо статичных классов объекты могут динамически изменять свои прототипы или добавлять новые методы/свойства во время выполнения.

3. Делегирование

Если объект не содержит запрашиваемый метод или свойство, он делегирует запрос своему прототипу (цепочка прототипов). Это заменяет классическое наследование "класс-подкласс".

4. Отсутствие классов

В чистом прототипном ООП нет разделения на классы и экземпляры. Все сущности — объекты.

Язык Self - пионер прототипного ООП

- **Разработчики :**
Self создавался в Стэнфордском университете группой исследователей, включая Дэвида Унгера (David Ungar) и Рэндалла Смита (Randall Smith) . Позднее к проекту присоединились сотрудники Sun Microsystems.
 - **Цель :**
Исследовать чисто объектно-ориентированную модель, где всё является объектом, а наследование реализуется через **прототипы** , а не классы.
 - **Первая версия :** 1987 год (Self 1.0), но популярность пришла с версией Self 3.0 (1992), которая ввела оптимизацию производительности через JIT-компиляцию .
-

Язык Self: ключевые концепции

1. Прототипное наследование

В Self нет классов. Объекты создаются клонированием существующих объектов (прототипов). Например:

```
self Копировать  
1 (| parent* = traits point |) // Создание объекта на основе p
```

Любой объект может быть прототипом для другого, что позволяет динамически изменять поведение.

2. Чистая объектная модель

Всё в Self — объект, включая числа, функции и даже операторы. Например, сложение `2 + 3` эквивалентно вызову метода `2.add(3)`.

3. Динамическая типизация

Типы объектов определяются во время выполнения, а не на этапе компиляции.

4. Слоты (Slots)

Объекты состоят из **слотов** — пар "имя-значение". Слоты могут хранить данные, методы или ссылки на другие объекты.

Язык Self: особенности

- **Делегирование :**

Если объект не содержит нужного метода, запрос перенаправляется его прототипу (механизм делегирования).

- **Метапрограммирование :**

Объекты могут модифицировать себя и свои прототипы во время выполнения.

Язык JavaScript

JavaScript

JavaScript был создан Бренданом Айком (Brendan Eich) в 1995 году во время его работы в компании Netscape Communications.

Влияние других ЯП:

- **Java** – C-подобный синтаксис
- **Scheme** – функции как объекты первого класса (можно передавать как аргументы), поддержка замыканий
- **Self** – прототипное наследование (вместо классов), динамическая типизация

Среды выполнения JavaScript

1. Браузеры

Самая распространенная среда для выполнения JavaScript.

- Движки :
 - V8 (Chrome, Edge, Opera, Brave)
 - SpiderMonkey (Firefox)
 - JavaScriptCore (Safari)
- Особенности :
 - Работа с DOM, событиями, AJAX, Web APIs (WebGL, WebSockets, localStorage).
 - Используется для создания интерактивных веб-страниц.

Среды выполнения JavaScript

2. Node.js

Серверная среда выполнения на основе движка V8 .

- Особенности :
 - Работа с файловой системой, сетевыми запросами, базами данных.
 - Используется для backend-разработки, создания API, микросервисов.
 - Пакетный менеджер `npm` или `yarn` для управления зависимостями.

JavaScript-код выполняется практически везде: от микроконтроллеров до серверов и браузеров. Это стало возможным благодаря гибкости языка и развитию движков (например, V8). Каждая среда использует свои API и инструменты, но сам язык остается единым стандартом (ECMAScript).

Объекты в JavaScript

Способы создания объектов:

- Объектный литерал
- Фабричная функция
- Фабричная функция и механизм прототипирования
- Метод `Object.create`
- Оператор `new` (функция-конструктор)
- Классы и конструкторы

Объектные литералы

В JavaScript объекты можно создавать с помощью литералов объектов (object literals), без использования классов или конструкторов.

```
javascript
1  const user = {
2    name: "Alice",
3    age: 30,
4    isAdmin: true,
5    hobbies: ["reading", "music"],
6    address: {
7      city: "Berlin",
8      country: "Germany"
9    }
10 };
```

Объектные литералы

В JavaScript объекты можно создавать с помощью литералов объектов (object literals), без использования классов или конструкторов.

```
javascript
1  const calculator = {
2    sum(a, b) {
3      return a + b;
4    },
5    multiply: function(a, b) {
6      return a * b;
7    }
8  };
9
10 console.log(calculator.sum(2, 3)); // 5
```

Сокращенная запись объектного литерала

```
let name = 'Иван',  
    age = 25;  
  
// Сокращенная запись объектного литерала  
const obj1 = {name: name, age: age}  
const obj2 = {name, age}
```

Деструктуризация обьекта

```
// Деструктуризация по обьекту
const obj3 = {name: 'Мария', age: 20};

name = obj3.name;
age = obj3.age;

({name, age} = obj3)

let {name: name2, age: age2} = obj3;
```

Spread-оператор

```
// spread-оператор
const o1 = {a: 1, b: 2}
const o2 = {c: 3, d: 4, e: 5}
const o3 = {...o1, ...o2}
```

Создание объектов с помощью фабричной функции

Фабричная функция

```
1 ✓ function createUser(name, age) {  
2   return {  
3     name,  
4     age,  
5     sayHello() {  
6       return `Привет, я ${this.name}, мне ${this.age} лет.`;  
7     }  
8   };  
9 }  
10  
11 const user1 = createUser("Алиса", 30);  
12 console.log(user1.sayHello()); // "Привет, я Алиса, мне 30 лет."
```

Инкапсуляция данных через замыкание

```
function createEmployee(name, salary) {
  return {
    getName: () => name,
    raiseSalary: (percent) => {
      | salary *= 1 + percent / 100;
    },
    getSalary: () => salary,
  };
}

const bond = createEmployee("James Bond", 100000);
console.log(bond.getSalary());

console.log({ bond });
```

- Инкапсуляция : Данные (`name` , `salary`) скрыты от внешнего мира.
- Публичный интерфейс : Только методы `getName` , `getSalary` , `raiseSalary` доступны для взаимодействия.
- Защита данных : Невозможно изменить `salary` иначе как через метод `raiseSalary` .

Проблема дублирования методов

Каждый созданный объект содержит отдельную копию методов, что приводит к избыточному использованию памяти.

```
function createEmployee(name, salary) {  
  return {  
    name,  
    salary,  
    // Методы дублируются в каждом объекте  
    getName() { return name; },  
    getSalary() { return salary; },  
    raiseSalary(percent) { salary *= 1 + percent / 100; }  
  };  
}  
  
const emp1 = createEmployee("Alice", 1000);  
const emp2 = createEmployee("Bob", 2000);  
  
console.log(emp1.getName === emp2.getName); // false (разные
```

Прототипы в JavaScript

Свойство `[[Prototype]]`

- Каждый объект в JavaScript имеет скрытую ссылку `[[Prototype]]`, которая:
 - Либо ссылается на другой объект (прототип),
 - Либо равна `null` (у объекта нет прототипа).
- Это свойство не доступно напрямую, но его можно изменять и читать через специальные методы.

Цепочка прототипов

Когда вы обращаетесь к свойству объекта, JavaScript:

1. Ищет его в самом объекте.
2. Если не находит — идет по ссылке `[[Prototype]]` и проверяет прототип.
3. Продолжает подниматься по цепочке, пока не найдет свойство или не достигнет `null`.

Пример:

```
javascript Копировать
1 const animal = {
2   legs: 4,
3   sound() {
4     console.log("Звук животного");
5   }
6 };
7
8 const dog = Object.create(animal); // dog.[[Prototype]] = animal
9 dog.name = "Рекс";
10
11 console.log(dog.legs); // 4 (наследуется от animal)
12 dog.sound(); // "Звук животного" (наследуется от animal)
```

Получение/изменение [[Prototype]]

- Чтение :

```
javascript
```

```
1 const proto = Object.getPrototypeOf(obj); // Получить [[Prototype]]
2 // или устаревший способ:
3 const proto = obj.__proto__;
```

- Изменение :

```
javascript
```

```
1 Object.setPrototypeOf(obj, newProto); // Изменить [[Prototype]]
2 // или устаревший способ:
3 obj.__proto__ = newProto;
```

Получение/изменение [[Prototype]]

```
const a = { n: 1 };  
const b = { m: 2 };  
  
Object.setPrototypeOf(a, b);  
  
const c = Object.getPrototypeOf(a);  
  
console.log(c);  
console.log(a.m);
```

Фабричная функция и прототипы

```
const employeePrototype = {
  raiseSalary(percent) {
    this.salary *= 1 + percent / 100;
  },
};

function createEmployee(name, salary) {
  const result = { name, salary };
  Object.setPrototypeOf(result, employeePrototype);

  return result;
}

// // -----
const bond1 = createEmployee("James Bond", 100000);
const bond2 = createEmployee("John Bond", 45000);

console.log({ bond1 });
console.log({ bond2 });
```

Создание объектов через Object.create

Object.create() — это метод в JavaScript, который позволяет создавать объекты с явно заданным прототипом.

javascript

```
1 v const animal = {
2   legs: 4,
3 v   sound() {
4     console.log("Звук животного");
5   }
6 };
7
8 // Создаем объект с прототипом animal
9 const dog = Object.create(animal);
10 dog.name = "Рекс";
11
12 console.log(dog.legs); // 4 (наследуется от animal)
13 dog.sound();          // "Звук животного" (наследуется от animal)
```

Оператор new (функция-конструктор)

```
function Employee(name, salary) {
  this.name = name;
  this.salary = salary;
  // return {a: 1, b: 2};
}

Employee.prototype.raiseSalary = function (percent) {
  this.salary *= 1 + percent / 100;
};

bond = new Employee("James Bond", 100000);
```

1. `new` создает новый пустой объект `{}`
2. `this` связывается с этим объектом
3. В этом объекте `[[Prototype]]` -> `Employee.prototype`
4. `Employee.prototype.constructor = Employee`
5. Запускается функция-конструктор `Employee`, в которой через `this` могут инициализироваться свойства объекта.
6. Объект, на который указывает `this`, автоматически возвращается из функции-конструктора и присваивается переменной.

Создание объектов оператором new

```
javascript 📄 ⬇  
1 ✓ function Person(name, age) {  
2     this.name = name;  
3     this.age = age;  
4 }  
5  
6 // Добавляем метод в prototype  
7 ✓ Person.prototype.greet = function() {  
8     console.log(`Привет, я ${this.name}, мне ${this.age} лет.`);  
9 };  
10  
11 // Создаем объект с new  
12 const alice = new Person("Алиса", 30);  
13 alice.greet(); // "Привет, я Алиса, мне 30 лет."
```

Создание объектов оператором new

Ключевые моменты

1. Связь с `prototype` :

- Созданный объект наследует методы из `Person.prototype` :

```
javascript 📄 ⬇  
1 console.log(alice.__proto__ === Person.prototype); // true
```

2. Отсутствие `new` :

- Без `new` функция-конструктор вернет `undefined` (или другой объект, если он явно возвращается), а `this` будет ссылаться на глобальный объект (например, `window` в браузере):

```
javascript 📄 ⬇  
1 const bob = Person("Боб", 25); // НЕ использовать new!  
2 console.log(bob); // undefined  
3 console.log(window.name); // "Боб" (глобальная переменная!)
```

Создание объектов с помощью class

Классы в ES6 — это удобный синтаксис (синтаксический сахар) для работы с **функциями-конструкторами** и **прототипами**.

Они не добавляют новых возможностей, но делают код более читаемым и похожим на классы в других языках (например, Java или Python).

Создание объектов с помощью class

Классы компилируются в функции-конструкторы с методами в `prototype`.

Пример класса:

```
javascript
1 class User {
2   constructor(name) {
3     this.name = name;
4   }
5
6   greet() {
7     console.log(`Привет, ${this.name}!`);
8   }
9 }
10
11 const user = new User("Алиса");
12 user.greet(); // "Привет, Алиса!"
```

Создание объектов с помощью class

Тот же код через функцию-конструктор:

```
javascript
1 function User(name) {
2   this.name = name;
3 }
4
5 User.prototype.greet = function() {
6   console.log(`Привет, ${this.name}!`);
7 };
8
9 const user = new User("Алиса");
10 user.greet(); // "Привет, Алиса!"
```

Что происходит под капотом:

- `class User` становится функцией-конструктором.
- Методы класса (например, `greet`) добавляются в `User.prototype`.

Создание объектов с помощью class

```
class Employee {
  constructor (name, salary) {
    |   this.name = name;
    |   this.salary = salary;
  };

  raiseSalary(percent) {
    |   this.salary *= 1 + percent / 100;
  };
}

bond = new Employee('James Bond', 100000);
console.log({bond});

bond.raiseSalary(10);
console.log({bond});
```

Геттеры и сеттеры

Геттеры (`get`) и сеттеры (`set`) в JavaScript — это специальные методы, которые позволяют определять поведение при чтении (`get`) или записи (`set`) свойств объекта. Они обеспечивают контроль над доступом к данным и позволяют реализовать инкапсуляцию.

1. Основная идея

Геттеры и сеттеры позволяют:

- Скрывать внутреннюю логику работы с данными .
- Добавлять проверки при установке значений.
- Вычислять значения динамически при их чтении.

Геттеры и сеттеры

```
1 class User {
2   constructor(name) {
3     this._name = name;
4   }
5
6   get name() {
7     console.log("Получение имени");
8     return this._name;
9   }
10
11  set name(newName) {
12    console.log("Установка имени");
13    if (newName.length > 0) {
14      this._name = newName;
15    } else {
16      console.error("Имя не может быть пустым!");
17    }
18  }
19 }
20
21 const user = new User("Алиса");
22 console.log(user.name); // "Получение имени", затем "Алиса"
23 user.name = "Боб";      // "Установка имени"
24 console.log(user.name); // "Получение имени", затем "Боб"
25 user.name = "";        // "Имя не может быть пустым!"
```

Геттеры и сеттеры

```
class Person {
  constructor(last, first) {
    |   this.last = last;
    |   this.first = first;
  };

  get fullName() { return `${this.last}, ${this.first}` };

  set fullName(value){
    |   // const names = value.split(' ');
    |   // this.first = names[0];
    |   // this.last = names[1];

    |   [this.first, this.last] = value.split(' ');
  };
}

const bond = new Person('Bond', 'James');
const name = bond.fullName;
console.log({name});

bond.fullName = 'John Dow';

console.log({bond});
```

Приватные свойства

1. Использование соглашений (имена с подчёркиванием)

Это не настоящая инкапсуляция, а соглашение между разработчиками. Свойства с префиксом `_` считаются "приватными", но технически они остаются доступными.

```
1 class User {
2   constructor(name, age) {
3     this._name = name; // "Приватное" свойство
4     this._age = age;
5   }
6
7   get name() {
8     return this._name;
9   }
10
11  set name(newName) {
12    if (newName.length > 0) {
13      this._name = newName;
14    } else {
15      console.error("Имя не может быть пустым!");
16    }
17  }
18 }
19
20 const user = new User("Алиса", 30);
21 console.log(user._name); // "Алиса" (технически доступно)
22 user._name = "Боб";      // Можно изменить напрямую
23 console.log(user._name); // "Боб"
```

Приватные свойства

4. Приватные поля с # (ES2022+)

С появлением ES2022 в JavaScript появились нативные приватные поля, обозначаемые символом #. Они обеспечивают реальную защиту данных.

```
1 class User {
2   #name; // Приватное поле
3   #age;
4
5   constructor(name, age) {
6     this.#name = name;
7     this.#age = age;
8   }
9
10  getName() {
11    return this.#name;
12  }
13
14  setName(newName) {
15    if (newName.length > 0) {
16      this.#name = newName;
17    } else {
18      console.error("Имя не может быть пустым!");
19    }
20  }
21
22  getAge() {
23    return this.#age;
24  }
25 }
26
27 const user = new User("Алиса", 30);
28 console.log(user.getName()); // "Алиса"
29 user.setName("Боб");
30 console.log(user.getName()); // "Боб"
31 console.log(user.#name); // SyntaxError: Private field '#name' must be declared in ;
```

Наследование

```
class Employee {
  constructor (name, salary) {
    |   this.name = name;
    |   this.salary = salary;
  };

  raiseSalary(percent) {
    |   this.salary *= 1 + percent / 100;
  };
}

class Manager extends Employee {
  constructor (name, salary, bonus) {
    |   super(name, salary);
    |   this.bonus = bonus;
  };

  getSalary() { return this.salary + this.bonus }
}

const employee = new Employee('James Bond', 100000);
console.log({employee});

const manager = new Manager('John Doe', 200000, 50000);
console.log({manager});
```

Наследование

```
// Конструктор для Employee
function Employee(name, salary) {
    this.name = name;
    this.salary = salary;
}

// Метод raiseSalary добавляем в прототип Employee
Employee.prototype.raiseSalary = function (percent) {
    this.salary *= 1 + percent / 100;
};

// Конструктор для Manager
function Manager(name, salary, bonus) {
    // Вызываем конструктор Employee через call
    Employee.call(this, name, salary);
    this.bonus = bonus;
}

// Наследуем прототип Employee для Manager
Manager.prototype = Object.create(Employee.prototype);
Manager.prototype.constructor = Manager;

// Добавляем метод getSalary в прототип Manager
Manager.prototype.getSalary = function () {
    return this.salary + this.bonus;
};
```



Class Expression

Class Expression в JavaScript — это способ создания класса как выражения, которое можно присвоить переменной, передать в функцию или использовать в любом месте, где допустимы выражения. В отличие от **Class Declaration** (объявление класса через `class Имя {}`), Class Expression более гибкий и позволяет создавать классы динамически.

1. Анонимный Class Expression

```
javascript
1  const User = class {
2    constructor(name) {
3      this.name = name;
4    }
5
6    sayHello() {
7      console.log(`Hello, ${this.name}!`);
8    }
9  };
10
11 const user = new User('Alice');
12 user.sayHello(); // Hello, Alice!
```

Class Expression

2. Именованный Class Expression

Имя класса доступно только внутри самого класса (например, для рекурсивных вызовов):

```
javascript 📄 ⬇  
1 const Counter = class CounterClass {  
2   constructor() {  
3     this.count = 0;  
4   }  
5  
6   increment() {  
7     this.count++;  
8     if (this.count < 5) {  
9       CounterClass.prototype.increment.call(this); // Рекурсивный вызов  
10     }  
11   }  
12 };  
13  
14 const counter = new Counter();  
15 counter.increment();  
16 console.log(counter.count); // 5
```